Analyse des Programmes et Sémantique
TD1 compléments

Basile Starynkevitch

janvier-avril 2013

http://starynkevitch.net/Basile basile@starynkevitch.net



A propos des arbres 1

L'égalité des arbres se programme récursivement :

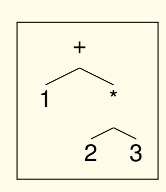
```
static bool equalTree(Tree a1, Tree a2) {
  if (a1==a2) return true;
  if (a1 instanceof Leaf && a2 instanceof Leaf)
     return a1.content() == a2.content();
  else if (a1 instanceof Node && a2 instanceof Node) {
     if (((Node) a1).oper() != ((Node) a2).oper()) return false;
     int ar = ((Node) a1).arity(); // == a2.arity()
     for (int ix=0; ix<ar; ix++)
        if (!equalTree(((Node) a1).son(ix), ((Node) a2).son(ix)))
          return false;
     return true;
  }
  else return false;                    }
```

L'ordre dans lequel on récure et compare les fils pourrait être différent :

```
for (int ix=ar-1; ix>=0; ix-)
```

A propos des arbres 2

Si on teste l'égalité par rapport à un arbre donné et figé, on peut avoir intérêt à ordonner les tests sur mesure ; par exemple pour tester l'égalité d'un expression arithetique quelconque `e` à `1+(2*3)` :




```
static bool eq1p2x3(Tree e) {
  if (e instanceof Node && ((Node)e).oper() == OP_PLUS) {
     Tree e1 = ((Node)e).son(1);
     if (e1 instanceof Node && ((Node)e1).oper() == OP_MULT) {
        Tree e0 = ((Node)e).son(0);
        Tree el = ((Node)e1).son(0);
        Tree er = ((Node)e1).son(1);
        if (e0 instanceof Leaf && e1 instanceof Leaf && e2 instanceof Leaf)
          if (e0.content() == 1 && e1.content() == 2 && e2.content() == 3)
            return true;
     } }
  return false; }
```

Arbres troués

Un trou est noté $\Box$ et peut être rempli avec un [sous-] arbre.

L'expression `1+(2*3)` est compatible avec l'arbre troué `1+`$\Box$ c.à.d. + (1, $\Box$) et aussi avec l'arbre troué $\Box$`+(`$\Box$ `*` $\Box$`)`

- chaque non-terminal ν se voit étendre d'une règle de production $\nu \to ...|\Box$
- en toute rigueur, chaque non-terminal (i.e. `Instr` ou `Expr`) aurait son trou

La compatibilité d'un arbre avec un arbre troué se code récursivement
Si le trou était représenté par `null` il suffirait d'ajouter au début du code de `equalTree` le test **`if`** `(a2==`**`null`**`)` **`return true`**`;` pour obtenir `compatibleTreeHoled`

Note : Si on voulait tester la compatibilité d'un arbre quelconque *a* avec trois arbres troués $T_1\, T_2\, T_3$ on pourrait "factoriser" des tests pour les parties communes.

Motifs ("patterns" ou patrons)

On généralise le trou $\square$ en des **variables de motif** notées [1] $\mathring{V}, \mathring{W}, \ldots$

Un **motif** est "un arbre avec des variables de motif" par exemples $1+\mathring{A}$ ou $\mathring{A}+(\mathring{B}*\mathring{C})$

Filtrer (anglais "match") un arbre par un motif (semi-unification) :

- le filtrage peut *échouer*
- s'il réussit, le filtrage instancie les variables du motif par des sous-arbres

Si une même variable apparait *plusieurs fois* dans le motif on dit que le motif est *non-linéaire*, et il faut alors tester l'égalité de sous-arbres.
Certains languages de programmation (Ocaml, Scala ...) offrent le filtrage par un motif. [2]

Voir `http://fr.wikipedia.org/wiki/Filtrage_par_motif`

---

1. C'est une notation personnelle ; chacun a la sienne
2. Les motifs y sont une classe syntaxique à part, comme le sont les expressions et instructions.

Applications des motifs

- les expressions regulières (regexp) d'Emacs (notations `\1` et `\&`)
- filtrage des trames TCP/IP
- générateurs de programmes d'analyse syntaxique (*ANTLR*, `bison`)
- analyses statiques dirigées par la syntaxe abstraite
- la simplification des expressions arithmétiques se formalise par des ré-écritures dirigées par des motifs (pattern directed rewriting), par exemples $0 + \mathring{X} \rightarrow \mathring{X}$ et $\mathring{X} * 0 \rightarrow 0$
  La simplification des expressions [3] formées des nombres rationnels, des constantes irationnelles e et π, d'une variable x, des quatre operations $+ - \times /$, des fonctions trigonométriques *sin* et *cos*, logarithmes *ln* et exponentielle *exp* est **indécidable** `http://fr.wikipedia.org/wiki/Théorème_de_Richardson`

3. Comme celles d'un exercice de 1re ou Terminale au lycée



Digression : l'Intelligence Artificielle [Classique] 1

- à l'origine (A.Turing 1950, J.MacCarthy 1960, J.Pitrat 1966, A.Colmerauer 1972, J-L.Laurière 1977, D.Lenat, R.Schank 198x ...) un fantasme fécond : faire des programmes qui se comportent aussi intelligemment q'un être humain
- les systèmes experts formalisent des connaissances empiriques floues

  IF the identity of the germ is not known with certainty AND the germ is gram-positive AND the morphology of the organism is "rod" AND the germ is aerobic THEN there is a strong probability that the germ is of type enterobacteriacae

- les systèmes experts ont trop [4] promis
  `http://en.wikipedia.org/wiki/AI_winter`
- aujourd'hui l'IA se réduit souvent a une approche algorithmique des traitements symboliques (ou non-numériques)

---

4. Note personnelle : Peut-être que de nos jours l'analyse statique des programmes par interprétation abstraite "promet" bientôt trop ?

Digression : l'Intelligence Artificielle [Classique] 2

- connaissances procédurales (nos programmes) $\neq$ connaissances déclaratives
  une connaissance est déclarative si elle ne donne pas son mode d'emploi :
  “l'article s'accord en genre et en nombre avec le nom”
- l'I.A. classique a eu le mérite de s'attaquer à des problème mal posés (diagnostic médical ou technologique, pilotage de haut fourneau [Sachem])
- systèmes experts : base de faits (milliers, millions), base de règles (centaines, milliers), le moteur d'inférence
- règles de production (en vrac) : `SI` *conditions* `ALORS` *actions* (ou *conséquents*)
  Idéalement, l'ordre des règles, et des conditions et des conséquents est sans importance !
- les faits et les conditions, les conséquents sont formalisés en prédicats logiques avec variable

Digression : Systèmes experts et au delà

Les formes de raisonnement automatiques

- chaînage avant : on modifie la base de faits en appliquant les règles $\Rightarrow$ pour ajouter des nouveaux faits (Snark, CLIPS)
- chaînage arrière : on travaille avec des buts (et des sous-buts) qu'on essaie de prouver (Prolog)
- chaînage mixte

(certains ont arnaqués leur client en faisant croire qu'un expert peut facilement expliciter sa connaissance en des règles)

approche **méta** : fournir déclarativement des **métaconaissances** qui synthétisent les programmes pour traiter les connaissances "l'IA est un problème trop difficile pour l'homme, seule l'IA peut y arriver" (J.Pitrat) bootstrap

Récréation

Lectures récréatives suggérées :

- Douglas Hofstadter : *Godel, Escher, Bach* 1979
  `http://en.wikipedia.org/wiki/Gödel,_Escher,_Bach`
- Jean-Louis Laurière : *Intelligence Artificielle* 1987
- Jacques Pitrat : *Méta-connaissances (futur de l'IA)* 1990
- D.Hofstadter : *I am a Strange Loop* 2007
- J.Pitrat : *Artificial Beings, the conscience of a conscious machine* 2009
- M. Minsky *Society of Mind* 1988
  `http://en.wikipedia.org/wiki/Society_of_Mind`
- D.Lenat et al. *Eurisko*, *CyC*, ... `http://en.wikipedia.org/wiki/Cyc`, `www.opencyc.org`

l'Unification intuitivement

On unifie (symétriquement) deux motifs ou deux arbres avec *variables* ; l'unification échoue ou réussit en substituant des variables

$$1+(2*X) \approx Y+(2*3)$$



Ces deux termes s'unifient avec les substitutions $X \to 3$ & $Y \to 1$

http://fr.wikipedia.org/wiki/Unification

http://en.wikipedia.org/wiki/Unification_(computer_science)

http://en.wikipedia.org/wiki/Occurs_check $X \approx X + 1$?

NB : l'algorithme d'unification produit et travaille avec un environment de substitution de variables (et peut échouer)

Un peu de Prolog

Variables (ECLIPSE-CLP) en majuscules [5], atomes constants et prédicats en minuscules
Base de faits (prédicats) :

- **`man(eric).`** signifie "`eric` (un symbole atomique) est un homme"
- **`married(eric, christiane).`** signifie : "Eric est marié à Christiane"

Chargement de la base (de faits et de règles) **`[family].`** pour charger le fichier `family.ecl`
Interrogation par la requête **`?- man(X).`** qui donne plusieurs résultats successifs (retour-arrière, backtrack)
Base de règles (chaînage arrière) :

```
sibling(X, Y) :- father(F, X), father(F, Y), X \= Y.
```

càd pour prouver `sibling(X,Y)` ("`X` est frère ou soeur de `Y`") prouver `father(F, X)` et puis `father(F, Y)` et puis `X \= Y`
NB : Prolog est peu déclaratif : l'ordre des règles et des prédicats y est [trop] important !

5. D'autres Prolog ont eu d'autres conventions, comme `=x` ou `?x` pour une variable...

Lancer **`eclispe-clp`** en ligne de commande.

